Application Modes: A Narrow Interface for End-User Power Management in Mobile Devices

Marcelo Martins Rodrigo Fonseca
Brown University

ABSTRACT

Achieving perfect power proportionality in current mobile devices is not enough to prevent users from running out of battery. Given a limited power budget, we need to control active power usage, and there needs to be a prioritization of activities. In the late 1990s, Flinn and Satyanarayanan showed significant energy savings using a concept of data fidelity to drive mobile application adaptation, informed by the battery lifetime desired by the user and the OS's evaluation of energy supply and demand. In this paper we revisit and expand this approach, recognizing that with current hardware there are even higher potential savings, and that increased diversity in applications, devices, and user preferences requires a new way to involve the user to maximize their utility. We propose Application Modes, a new abstraction and a narrow interface between applications and the OS that allows for a separation of concerns between the application, the OS, and the user. Application Modes are well suited to eliciting user preferences when these depend on multiple dimensions, and can vary between users, time, and context. Applications declare modes – bundles of functionality for graceful degradation when resource-limited. The OS uses these modes as the granularity at which to profile and predict energy usage, without having to understand their semantics. It can combine these predictions with application-provided descriptions, exposing to the user only the high-level trade-offs that they need to know about, between battery lifetime and functionality.

1. INTRODUCTION

Battery life has been a fundamental limitation in mobile devices for as long as they have existed, despite a vast body of literature on power management extending back almost two decades (§6). In fact, increasingly demanding applications greatly exceed the average power draw that would be required for batteries to last through a typical charging period [9, 25].

There is a wide spectrum of proposed solutions for power management. A first class of solutions deals with the management of idle-resource power, by automatically switching hardware components to low-power states when not in use. These include timeout policies for hibernation, suspending disks, displays and radios; and CPU voltage and frequency scaling. The outcome, if these are perfect, is an energy-proportional system [5]. Although necessary, these are not sufficient to solve the increasing energy-deficit problem, because they have no effect when there is *active demand* for resources.

To reduce the active demand, there must be a prioritization of functionality. In the late 1990s, Ellis [8, 27] recognized that the offered workload has to be changed by user-driven prioritization and lifetime goals, and Flinn and Satyanarayanan established, with the Odyssey system, that application adaptation can provide substantial energy gains [10, 16]. In Odyssey, applications automatically and dynamically change their behavior to limit their energy consumption and achieve user-specified battery lifetime, guided by the operating system. The adaptation involves a trade-off between energy use and application data quality, which they called *fidelity*. Fidelity is application-specific and opaque to the OS. The role of the OS is to direct the adaptation based on its evaluation of the supply and demand of energy, and their relation to the expected battery duration. When the OS detects the lifetime goal as unachievable, it issues upcalls to applications so they reduce their fidelity. The user inputs two pieces of information: the desired lifetime, and a prioritization of applications to order their adaptation, whereas application developers are responsible for implementing different fidelity levels.

Flinn and Satyanarayanan were the first to simultaneously involve the OS, the applications, and the user in power management, and many factors in today's environment make it opportune to revisit and extend their approach, which we do in this paper. Due to a combination of more complex applications, multiple devices, and a diverse user base, in some cases there is no single fidelity metric that is common to all users in all contexts, making automated approaches to adapt some applications ineffective. Furthermore, given advances in hardware and in lower-level software (*e.g.*, ACPI), devices are much more efficient when idle, making higher-level approaches that reduce active demand much more effective now than a decade ago.

In [10], as in [12], a fundamental assumption is that there is a well-defined trade-off between fidelity (or QoS) and energy use. This means that an application developer knows the app configurations that lie in the *Pareto frontier* of this trade-off, enabling an automated algorithm to decide the state based on the available energy.

Even though this still holds for many applications, this is not always true. As we show in §2, two users with different preferences can have very different trade-offs between energy usage and *utility* from an application. The key observation is that in these cases, automated adaptation fails, and the runtime system must elicit preferences from the user. The main challenge is how to involve the user at the right time and *at the right level*. She should only worry about tangible aspects of the device operation, such as *lifetime* and *functionality*, and not be concerned with how these are implemented or achieved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ACM HotMobile'13, February 26–27, 2013, Jekyll Island, Georgia, USA.
Copyright 2013 ACM 978-1-4503-1421-3/13/02 ...$15.00.

In this paper we propose *Application Modes* (§4), an interface between applications and the OS that eases this communication. Rather than exposing a *metric*, applications declare to the OS one or more *modes*, which comprise reductions of functionality with presumed power savings. Modes carry a human-readable description of the resulting functionality, and the promise of switching when requested by the OS. Similarly to previous works [10, 2], we assume that the OS can predict how long the device will last with the application in each mode, and then request its change when appropriate. However, recognizing that different modes may have different utilities for different users, the decision of when to switch modes involves the user when necessary, by combining the description provided by the application with the predictions of change in lifetime provided by the OS.

2. MOTIVATION

In this section we use power measurements with two common smartphone applications — a navigation and a video-recording application — to illustrate two main points. First, we confirm and extend earlier findings by Flinn and Satyanarayanan [10] demonstrating how changes in application behavior can substantially affect energy consumption. Second, using the video-recording application, we show that different users can have very different Pareto frontiers in the utility-energy trade-off, making globally automated decisions ineffective to maximize utility.

We measure the power draw of running these applications in very different *modes*, or bundles of settings. We did our measurements on a Samsung Galaxy Nexus running CyanogenMod ICS 4.1.0. We measured the power draw of the entire phone connecting a Monsoon power monitor to the smartphone battery. To discriminate the energy consumed due to application, we first measured the energy consumed by the phone in the idle state, *i.e.*, not running any applications apart from the base system, and established two baselines with the screen on and off. We kept the screen brightness to the same level for all runs where the screen was on. For navigation, we downloaded data using the 3G data connection when needed, and for the recording application, we used the WiFi network for data upload.

Navigation System Turn-by-turn navigation exercises several hardware resources, including the CPU, GPU, audio, networking, and GPS. It is used in sometimes critical situations, when there is little battery left and the user is in need of orientation to arrive at her destination (and a charging opportunity). There are also interesting trade-offs in functionality, utility, and energy use, depending on which subset of resources the application uses.

We demonstrate potential savings from running Osmand[1] and Navfree[2], two turn-by-turn navigation applications for Android devices with online/offline features, and modifying their settings. We consider five modes, listed in Table 1, from selected parameters for the screen and audio outputs, map-data source and routing. These settings are not transparent to the user, and make specific trade-offs between accuracy and resource usage for a given route. Notable differences between settings include the use of previously downloaded vector maps instead of online tile data, disabling the display and using only audio for directions, and downloading directions for the user to write down! We compare the power draw of calculating and outputting the directions for a fixed, four-mile route.

Figure 1 shows, for each mode, the distribution of instantaneous power-draw samples from the device over the entire experiment. The "Full Features" mode yields a richer trajectory, including extra

[1]http://osmand.net

[2]http://www.navmii.com

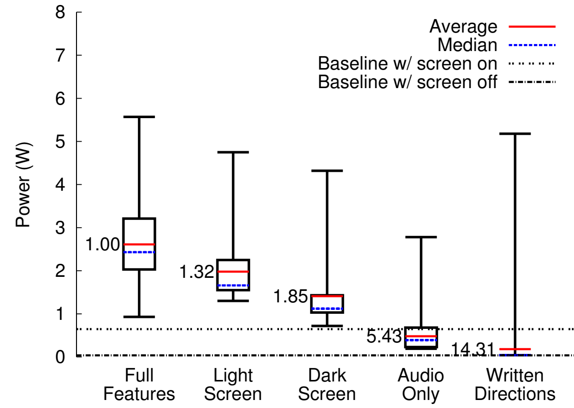

Figure 1: Power draw distributions for the navigation app in different output/routing settings (*cf.* Table 1). The vertical bars show the maximum and minimum power draw, and the boxes the 1st and 3rd quartiles, with the median and average indicated. The number to the left of each bar shows the improvement in battery usage relative to the "Full Features" scenario. Greater energy savings can be achieved by reducing the output quality.

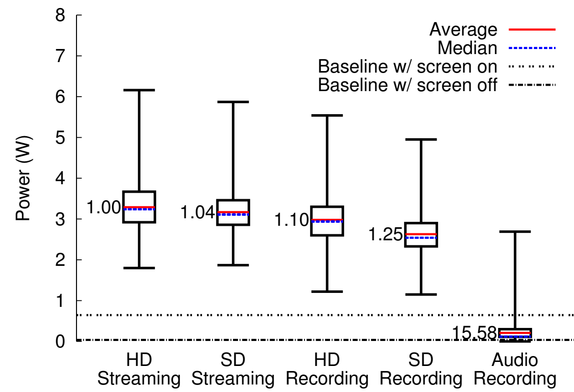

Figure 2: Power draw of different modes for the media-streaming app (*cf.* Table 2).

information like points of interest (POI), at the expense of a larger power profile. As we reduce the number of enabled settings we can see a drop in energy expenditure along with a decrease in the quality of routing information. The "Written Directions" mode draws on average more than 14× less power than "Full Features". The former's high variance stems from briefly using the screen and radio to search for directions; yet, its average power draw is much smaller than its counterparts. In exchange, the user has to take notes of the route before the trip and use them as the only source of information to reach the destination[3]. The potential savings are very significant, provided the user accepts the decrease in quality. In this example, like the ones in previous works, there is a total order in the utility of the modes that is likely agreeable to all users. As such, two alternatives for adaptation can work: as in Odyssey, if the OS knows the user's expected lifetime, the OS can request an increase or decrease in fidelity. Alternatively, we can use Application Modes to expose to users the functionality and expected lifetime of the device in each mode, for them to choose.

[3]This mode was motivated by one of the authors actually having had to do this one time!

Mode Name	Display Settings	Routing Settings	Program Used
Full	Online map tiles and overlays (Mapnik), POIs, compass, polygons	CloudMade routing (online)	Osmand
Light Screen	Offline vector maps, no POIs, no compass, polygons, day mode (light screen)	Osmand routing (offline)	Osmand
Dark Screen	Offline vector maps, no POIs, no compass, no polygons, night mode (dark screen)	Osmand routing (offline)	Osmand
Audio Only	Screen off	Navfree routing (offline)	Navfree
Written Directions	Screen on for browser search, off afterwards, no audio	Google Maps (online)	Browser

Table 1: Alternatives to navigating a four-mile course for the navigation app. Upper modes yield higher-quality routes in exchange for greater resource usage.

Mode Name	Encoding Settings	File size (MB)	Stream transmission?	Program Used
HD Streaming	720p video, hi-def audio	158.60	Live streaming via RTSP	LiveStream
HD Recording	720p video, hi-def audio	290.1	Upload when recharging	SpyCam
SD Streaming	480p video, med-def audio	47.16	Live streaming via RTSP	LiveStream
SD Recording	480p video, med-def audio	183.4	Upload when recharging	SpyCam
Audio Recording	Audio only, screen off	0.58	Upload when recharging	Sound Recorder

Table 2: Functionality alternatives for the media-streaming app, varying encoding quality and immediacy.

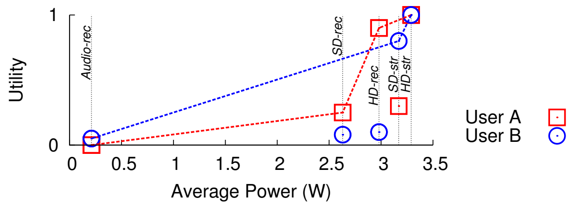

Figure 3: Utilities for the streaming modes for two users. 'A' prefers high-quality capture, whether streamed or recorded, whereas 'B' values immediacy over quality. The Pareto frontiers (the dashed lines) are different, and no single policy can choose between "SD Streaming" and "HD Recording" for all users.

Media Streaming Our second example is an audio and video streaming application which, as navigation, is widely used, leverages different hardware resources, and has interesting trade-offs.

We consider the power draw of capturing and transmitting a five-minute video feed using three similar applications. We performed measurements on different settings of the LiveStream, SpyCam[4] and MIUI Sound Recorder[5] apps using the aforementioned setup. We modify their functionality by selecting parameters for the video and audio encoding and for when to upload the captured media. We consider five basic sets of settings (Table 2), choosing whether to stream or record for later upload, and whether to encode video in high definition, standard definition, or audio only. Once again, these settings are not transparent to the user, and make specific trade-offs between quality and timeliness of the uploaded media.

Figure 2 shows the power draw of each mode. The "Audio Recording" mode draws on average over 15× less power than the "HD Streaming" mode. "Audio Recording" generates the least number of bytes, does not use the screen, camera, or video-encoding hardware, and does not include the transmission energy, as this is done only when the device is recharging.

Figure 3 shows the same modes, with a numerical utility for hypothetical users (which could even be the same user in different contexts). User 'A' is interested in obtaining high-quality video, whereas user 'B' values immediacy. The graph shows that the Pareto frontiers for the two users are very different, and that there is no consistent ordering of the modes, particularly between HD recording and SD streaming. This example highlights that neither the OS nor the application can know a priori the utility of the modes for each user. In this case, and in general when there are multiple dimensions that different users value differently, the automatic selection of a mode breaks down.

[4]http://dooblou.blogspot.com

[5]http://github.com/MiCode/SoundRecorder

3. THE USER NEEDS TO DRIVE

In this section we argue why the user, the applications, and the OS must all be involved in limiting the active demand of mobile devices to achieve maximum *value* out of a limited energy budget.

1. The OS cannot always know the resource priorities of all applications. If an application is consuming too much energy, the OS could limit the resources offered to it, such as CPU time, bandwidth, or access to precise location. Robust applications should sustain such reductions and adapt. However, such arbitrary reductions can be frustrating to the user, *as the value of different functionalities to her may be hard to predict*. This is exacerbated when there are alternative reductions. If the OS decides that a videoconferencing application is spending too much energy, it could reduce its CPU or network allocation, but cannot know which will lead to a more acceptable degradation to the user.

2. Applications cannot always know the functionality priorities of the end-user. Applications are in a better position than the OS to make such decisions, but they may still not know the user's preferences. As the video example in the previous section highlights, there may be no total ordering of the modes in an application, so that the application developer cannot determine the modes in the Pareto frontier for a specific user. In this case, it is only the user who can determine the relative value of the alternatives, as just knowing the user's desired battery lifetime is not sufficient to maximize the utility automatically.

3. Users should choose at the right level, trading off functionality versus lifetime. Although many existing systems could involve the user, most require too much knowledge at the wrong level of abstraction. The user should only have to know about high-level functionality and device lifetime, and not be concerned about which components of the phone even exist. A user wanting her battery to survive a 12-hour flight should not need to understand or even specify the screen brightness, CPU frequency, scheduling algorithm, or the WiFi data rate of her smartphone to fulfill her needs. The phone should hide these trade-offs from the user whenever possible. Popular solutions for end-user energy management are based on *components* rather than functionality, requiring the user to know the resource implications of turning off 3G, GPS, synchronization, or Bluetooth. Frameworks like Cinder [19], Koala [21], and Chameleon [14] have mechanisms to limit resource usage per application, but suffer from the same problem – they assume mobile users are likely to become system administrators or programmers of their devices. On the other hand, other frameworks limit themselves to a single knob, such as lifetime [10, 27] or a preference between performance and duration [1], but as we show, in some cases, this is not enough to maximize utility.

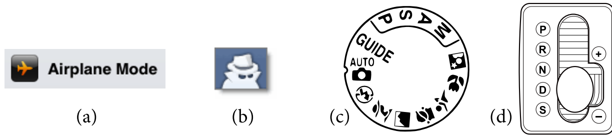

Figure 4: *Application Modes* abstract complex settings by a single, functionality-centered concept, and are common in diverse settings: (a) Airplane mode on phones, (b) Incognito mode in Chrome, (c) scene modes on a camera, and (d) driving modes on a semi-automatic transmission.

The only remaining question is why the OS should be involved at all, since applications could elicit users' preferences directly. The challenge here lies in the decision of when apps would offer these choices, as this requires knowledge of current and future energy availability. This functionality may require device-specific modeling, and should more naturally reside in the OS [6]. Requiring it in each app entails duplicated developer effort, and leads at best to poor or inconsistent results. The OS, on the other hand, is in the right position to provide an energy context to all apps, including profiling and predictions of energy usage and lifetime.

4. APPLICATION MODES

To address the concerns in the previous sections, we have implemented a new abstraction named *Application Modes*, bundles of functionality declared by applications to ease the separation of concerns required between applications, the user, and the OS for effective resource management. We borrow the concept of modes from several commonplace settings (see Figure 4), where they represent complex settings abstracted by a single functionality-centered, easy-to-understand concept. Application Modes resemble Chroma [2] in that very little application knowledge is exposed. Different from Chroma, users are not oblivious to the decisions made by the system, but actually have an active voice in making the decisions that affect their experience. Application Modes are particularly well suited to cases where there are multiple dimensions involved in the users' implicit preference function, with no total order among them, similarly to the different shooting modes on a camera, for example.

Power savings are achieved through *graceful degradation*. Developers create different modes for an application by selecting sets of functionalities that entail different application behaviors, as perceived by the user, in exchange for reduced energy consumption. Graceful degradation is achieved through various ways: different settings, different algorithms [20], even different programs.

The central part of the abstraction is a narrow interface between applications and the OS (see Listing 1). When opened for the first time, applications implementing this interface via a shared library declare to the OS their supported modes using a label and a user-friendly description of how each mode affects the user experience (the registerModes() system call). An OS-listening component intercepts this system call and saves in its database the metadata passed as arguments by the application, along with a unique identifier for each mode, and the currently selected mode. The latter is necessary to automatically restore the behavior of applications once they are reopened. Applications supporting our abstraction promise to switch to a given mode when instructed by the OS, whereas applications oblivious to this new API are not affected. Table 3 lists a few mode examples for different apps.

```
registerModes(List<ModeData>);   // system call
setMode(ModeId);                 // callback
```

Listing 1: API for data exchange between applications and the OS.

Application Modes represent a meaningful granularity at which the OS can do profiling of energy consumption, and make lifetime predictions for each one. Applications keep control of what resources to reduce in order to save power, but leave the decision of *when* to do so to the OS, which has detailed knowledge of the energy context of the entire device, and to the user, who can prioritize application functions based on her goals.

Battery lifetime and high-level functionality, metrics understandable by *both* users and developers, are used to guide adaptation. In one possible scenario, the OS notices that, at the current power draw, her phone will exhaust the battery before the next expected recharge. The OS then presents to the user a list of running apps, ordered by power draw. When the user selects an app from the list, the OS presents an interface similar to that in Figure 5, on which the user selects a different mode for the app, informed by its description (*i.e.*, functionality) and expected impact on battery lifetime. In another usage scenario, the user is presented with a notification of Application Modes support when opening a new program that implements the API. After clicking the notification, an interface similar to Figure 5 appears, and she explores the trade-off possibilities. Once a mode is selected, the OS instructs the application to change its settings using the setMode() callback function. It is the responsibility of the application developer to instruct her program to correctly change its behavior according to the mode selection.

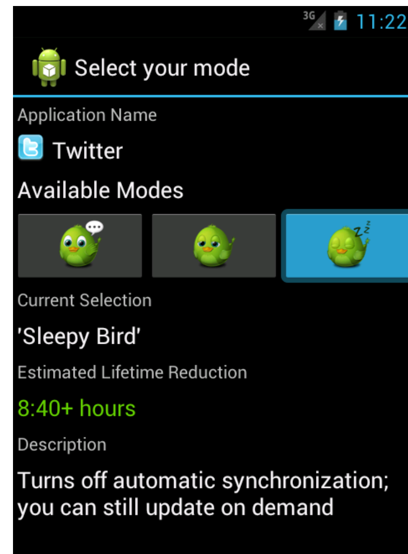

Figure 5: Interface to select application modes for a Twitter app.

5. CHALLENGES

For Application Modes to be adopted and maximize energy savings and user satisfaction, we and the community need to address a number of important challenges.

Energy Profiling and Forecasting The OS is at the right place to maintain an energy context for the device. This includes profiling the energy consumption due to apps, and forecasting the expected battery life given the current and alternative settings. Profiling at least at the granularity of Application Modes is needed for guiding developers on how to choose and optimize the modes themselves, and forecasting is key in telling the user the impact of choosing different modes. There has been significant progress in this area [10, 11, 15, 18, 24, 28], but there is still room for more precision, incorporation of segmented data from large user populations ([17] is a great start), and better support for sharing and delegating resource usage.

Developer Adoption Application Modes place a burden on developers, and are only useful if adopted. We are hopeful that there will be enough interest, given the benefits and the fact that some apps al-

Application	Full	Medium	Powersaver
Location Tracking	1m precision, real-time	1m precision, every 15 minutes	50m precision, at least once a day
Navigation	3D map, audio, real-time location	3D map, audio, location near turns	2D map with directions only
Video Upload	HD video, real-time upload	SD video, real-time upload	SD video, upload when charged
Twitter	Real-time updates	Updates every 5 minutes	Updates on demand

Table 3: Example modes for some applications.

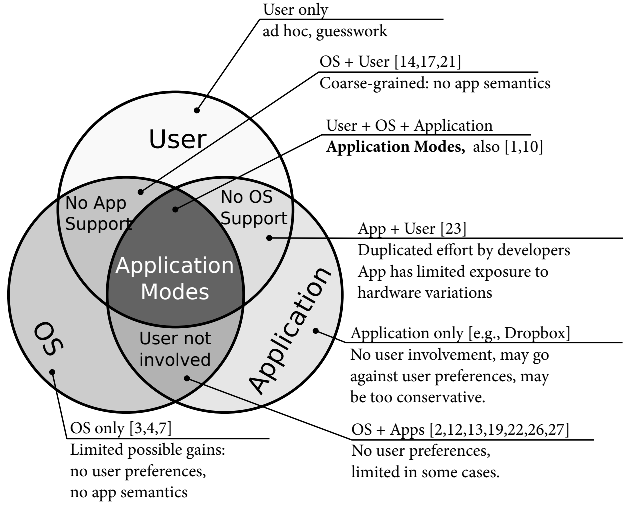

Figure 6: Design space of power-management solutions based on whether there is involvement or support from the user, the application, or from the OS. Application Modes elicit preferences from the user, using lifetime predictions from the OS, and functionality descriptions from the application.

ready change their behavior in response to the context. Dropbox on Android, for example, has an automatic photo-upload option, and disables the feature if the battery is low. With Application Modes, the developer would not have to write code to make sense of the battery context. As per the previous paragraph, developers need accessible energy profiling, as it is only by measuring the impact of design decisions that a developer will make informed decisions to effectively create modes. Lastly, the developer needs guidelines to not create too many modes, modes that do not affect perceived app behavior or expose too much detail. As in Chroma [2], we expect the number of *relevant* energy-saving optimizations from applications to be small.

Limiting User Involvement Changes to the end-user experience should be as little intrusive as possible. While user input is required in many situations, the OS and applications should autonomously act upon energy-related decisions as much as possible, through models, services, and profiles, only escalating what is really important.

Conflict Detection and Resolution Since most current mobile devices support multitasking, two or more apps could have conflicting modes. If two apps use the radio, having only one of them promising not to is of no use. The OS should have a mechanism to detect and resolve such conflicts, with possible involvement of applications and ultimately of end-users. Another source of conflicts are global settings not associated with any apps: a user setting the screen brightness is one example.

Other Resources The concept of Application Modes may apply to contexts other than energy, such as data usage and privacy. Related to previous challenges, the interaction and potential conflicts of these modes, and the possibility of an explosion of their number are important challenges we should address.

6. RELATED WORK

We build upon a large body of previous work that explores support from subsets of the user, the application, and/or the OS for power management. We structure our discussion around Figure 6, which lays out this space and points to representative works on each subset. Application Modes lie in the common intersection, judiciously involving applications, the OS, and the user when necessary.

Starting from OS-only support, ACPI-related and CPU voltage- and frequency-scaling techniques, along with TailEnder [3] and Cat-Nap [7], try to automatically determine a global machine behavior and estimate its best configuration in light of energy savings. In some specific occasions, they can be configured by users, although it is not always clear how these global settings will affect individual application functionality and energy-saving promises. Commercial OSes also have measures for reducing active demand, such as Windows Phone 7.5's Battery Saver Mode, which disables background processes and lowers the screen brightness when the battery charge drops below a certain level, or Apple iOS's disallowing of background processes as a global policy. As a research prototype, Llama [4] is an adaptive energy-management system for mobile devices that matches energy consumption with user behavior via probabilistic adaptation of system settings based on expected recharge time. Our work supplements Llama by integrating both user and applications in the adaptation process. In all these cases, because the OS lacks semantic knowledge of the application, the obtained savings are limited.

In the absence of OS and application support, users are forced to guess which settings, behaviors, or apps correlate with energy usage, having sometimes to understand specific implementation and hardware details. The OS can cooperate with the user by offering more visibility and obtaining hints about desired behavior, such as lifetime. We include here independent user-level services that monitor other running applications. Examples include Koala [21] and Carat [17]. Carat uses crowd-sourced usage data to suggest energy savings based on the device's state and past usage of similar devices, and presents the expected improvement in battery life if the user kills each of the currently running apps. Application Modes provide an interesting complement by increasing the granularity of possible user actions, as she can choose to change their functionality rather than just terminating them.

Some applications attempt to optimize their energy use without support from the OS or input from the user. Beyond development-time profiling (*e.g.*, [15]), some developers change the behavior of the app depending on the battery level, like the Dropbox example in §5. These applications may not have an obvious choice, though, without user input, when there are different changes in functionality in the Pareto frontier (*cf.* §2). Not surprisingly, some apps do involve the user. Sygic [23] is a voice-guided GPS navigation app that offers users distinct modes of operation based on different power-management profiles. While this approach is beneficial, not involving the OS has drawbacks. In the best case, there is a severe duplication of effort by independent developers, and in the worst case these apps will make suboptimal decisions, given the lack of coordination and the diversity of platforms on which they run.

Despite an initial effort to involve the user in energy-aware decisions [10], many recent solutions have focused on the energy-saving cooperation of the OS and app developers, perhaps to avoid alienating the user. A-States [12] are similar to Application Modes as they propose a narrow interface between apps and the OS to enable the coordination of power-saving measures without exchanging seman-

tic data. Our work differs in that we involve the end-user in the decisions, as the desirability of different modes, in our setting, is not monotonic, but can exhibit user-specific trade-offs. Eon [22] and Energy Levels [13] provide programming abstractions and runtimes to predict resource usage in wireless sensor networks (WSNs) and to automatically meet lifetime goals by deactivating or adapting parts of an application. Resource-usage prediction is facilitated by the single-threaded environment and periodic behavior of WSN applications. Users are likely the developers, who possibly understand the innards of such a complex system and there are almost no concerns about usability. Although Application Modes provide a similar abstraction and runtime, our focus is on multi-tasked, quasi-acyclic mobile applications that involves non-expert users.

Android's wakelocks also focus on this type of cooperation by allowing the kernel and user-space apps to control hardware suspending via reference counters to specific system components. To avoid races between suspend and wakeup events, all user-space processes need to be aware of the wakelocks interface. This is acceptable for Android, but not applicable to other Linux-based systems [26]. Developers have proposed alternatives to wakelocks for the mainline Linux kernel. Runtime Power Management[6] is similar to wakelocks, but is restricted to driver interaction with I/O devices. Autosleep[7] is a direct substitute which reuses components present in the mainline kernel. Applications could take advantage of both functionalities to cooperate with device drivers and provide hints on when to suspend.

There are very few works that, like ours, involve support from the three camps. Aside from Flinn and Satyanarayanan's energy-related extensions to Odyssey [10], Ghosts in the Machine [1] suggest giving the OS more visibility on the power state of I/O devices. Applications are adapted to issue hints to the device's power manager about their execution and help set the right power state, resulting in performance and energy-conservation improvements. Users express fidelity desires using a single unit-less knob that prioritizes performance or energy savings. While it is fundamental to limit the amount of input from the user, the diversity of mobile apps in our context makes this single dimension too restrictive to express user preferences, as different users will have preferences in the Pareto frontier that are not obvious to the developer or to the OS.

7. CONCLUSION

Application Modes enable the cooperation of the OS, applications, and users for efficient energy management in mobile devices. The user's only concerns are about differences in functionality and their impact on battery life. Application Modes are particularly well suited to cases in which there are multiple dimensions involved in the users' implicit preference function, with no total order among them. Applications provide the OS with discrete modes that express graceful degradation in face of limited energy. Further, the OS centralizes all of the knowledge and models about the current and future energy contexts, removing this burden from apps. We plan to use our abstraction and interface prototype for Android to instrument more applications, and conduct full-platform power measurements on real devices. The real measure of success for an interface is adoption, and we plan on conducting user studies with both developers and end-users.

Acknowledgments We thank Prabal Dutta, Srikanth Kandula, Deepak Ganesan, Shriram Krishnamurthi, Jonathan Mace, the anonymous reviewers, and our shepherd, Landon Cox, for their feedback. Marcelo was funded in part by a generous gift from Intel Corporation.

[6] http://lwn.net/Articles/347573

[7] http://lwn.net/Articles/479841

8. REFERENCES

[1] M. Anand, E. B. Nightingale, and J. Flinn. Ghosts in the machine: Interfaces for better power management. In *MobiSys'04*.

[2] R. K. Balan, M. Satyanarayanan, S. Y. Park, and T. Okoshi. Tactics-based remote execution for mobile computing. In *MobiSys '03*.

[3] N. Balasubramanian, A. Balasubramanian, and A. Venkataramani. Energy consumption in mobile phones: a measurement study and implications for network applications. In *IMC'09*.

[4] N. Banerjee, A. Rahmati, M. D. Corner, S. Rollins, and L. Zhong. Users and batteries: Interactions and adaptive energy management in mobile systems. In *Ubicomp'07*.

[5] L. A. Barroso and U. Hölzle. The case for energy-proportional computing. *Computer*, 40(12):33–37, 2007.

[6] D. Chu, A. Kansal, J. Liu, and F. Zhao. Mobile apps: It's time to move up to CondOS. In *HotOS'11*.

[7] F. R. Dogar, P. Steenkiste, and K. Papagiannaki. Catnap: Exploiting high bandwidth wireless interfaces to save energy for mobile devices. In *MobiSys'10*.

[8] C. S. Ellis. The case for higher-level power management. In *HotOS'99*.

[9] H. Falaki, R. Mahajan, S. Kandula, D. Lymberopoulos, R. Govindan, and D. Estrin. Diversity in smartphone usage. In *MobiSys'10*.

[10] J. Flinn and M. Satyanarayanan. Energy-aware adaptation for mobile applications. In *SOSP'99*.

[11] W. Jung, C. Kang, C. Yoon, D. Kim, and H. Cha. Appscope: Application energy metering framework for Android smartphone using kernel activity monitoring. In *USENIX ATC'12*.

[12] A. Kansal, J. Liu, A. Singh, R. Nathuji, and T. Abdelzaher. Semantic-less coordination of power management and application performance. In *HotPower'09*.

[13] A. Lachenmann, P. J. Marrón, D. Minder, and K. Rothermel. Meeting lifetime goals with energy levels. In *SenSys'07*.

[14] X. Liu, P. Shenoy, and M. D. Corner. Chameleon: Application-level power management. *IEEE Trans. on Mob. Comp.*, 7(8), 2008.

[15] R. Mittal, A. Kansal, and R. Chandra. Empowering developers to estimate app energy consumption. In *MobiCom'12*.

[16] B. D. Noble, M. Satyanarayanan, D. Narayanan, J. E. Tilton, J. Flinn, and K. R. Walker. Agile application-aware adaptation for mobility. In *SOSP '97*.

[17] A. J. Oliner, A. P. Iyer, E. Lagerspetz, S. Tarkoma, and I. Stoica. Carat: Collaborative energy debugging for mobile devices. In *HotDep'12*.

[18] A. Pathak, Y. C. Hu, and M. Zhang. Fine grained energy accounting on smartphones with eprof. In *EuroSys'12*.

[19] A. Roy, S. M. Rumble, R. Stutsman, P. Levis, D. Mazières, and N. Zeldovich. Energy management in mobile devices with the Cinder operating system. In *EuroSys'11*.

[20] M. Satyanarayanan and D. Narayanan. Multi-fidelity algorithms for interactive mobile applications. *Wirel. Netw.*, 7(6).

[21] D. C. Snowdon, E. Le Sueur, S. M. Petters, and G. Heiser. Koala: A platform for OS-level power management. In *EuroSys'09*.

[22] J. Sorber, A. Kostadinov, M. Garber, M. Brennan, M. D. Corner, and E. D. Berger. Eon: A language and runtime system for perpetual systems. In *Sensys'07*.

[23] Sygic. GPS navigation for smart phones. http://www.sygic.com.

[24] K. N. Truong, J. A. Kientz, T. Sohn, A. Rosezwieg, A. Fonville, and T. Smith. The design and evaluation of a task-centered battery interface. In *Ubicomp'10*.

[25] N. Vallina-Rodriguez, P. Hui, J. Crowcroft, and A. Rice. Exhausting battery statistics: Understanding the energy demands on mobile handsets. In *MobiHeld'10*.

[26] R. J. Wysocki. Technical background of the Android suspend blockers controversy. http://lwn.net/images/pdf/suspend_blockers.pdf.

[27] H. Zeng, C. S. Ellis, A. R. Lebeck, and A. Vahdat. ECOSystem: Managing energy as a first class operating system resource. In *ASPLOS-X'02*.

[28] L. Zhang, B. Tiwana, Z. Qian, Z. Wang, R. P. Dick, Z. M. Mao, and L. Yang. Accurate online power estimation and automatic battery behavior based power model generation for smartphones. In *CODES+ISSS'10*.